[redacted]

Report No. 471

PRUNING AND BRANCHING METHODS FOR DESIGNING OPTIMAL NETWORKS BY THE BRANCH-AND-BOUND METHOD

by

Tomoyasu NAKAGAWA
Hung Chi LAI
Saburo MUROGA

August 1971

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 471

PRUNING AND BRANCHING METHODS FOR DESIGNING OPTIMAL NETWORKS BY THE BRANCH-AND-BOUND METHOD*

Tomoyasu NAKAGAWA
Hung Chi LAI
Saburo MUROGA

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

* This work was supported in part by the National Science Foundation under Grant No. NSF GJ-503.

CONTENTS

INTRODUCTION

In this paper, we present a branch-and-bound algorithm for obtaining optimal NOR networks [3, 4]. This algorithm differs from Davidson's in the inclusion of the following two new features; i.e., (i) an efficient pruning of non-optimal networks and (ii) a different branching strategy for enumerating feasible networks. The feature of pruning non-optimal networks is based on the construction of better networks by eliminating redundant gates and connections of the networks which the enumerative part of the branch-and-bound algorithm generates; we call this feature the redundancy check The principle of our branching strategy is based on an attempt to minimize the width of the search tree of the branch-and-bound algorithm.

We present in section 1 the scheme of the redundancy check which we incorporated into our computer program of the branch-and-bound method, called ILLOD-(NOR-B) (ILlinois LOgical Design with NOR-gates by Branch-and-bound method).

In section 2, we describe the branching strategy. (The entire algorithm of the branch-and-bound method is presented in the paper [3].) For comparison, we summarize the differences in branching strategy between Davidson's algorithm and ours, and present some computational results with both versions.

Section 3 contains computational experiments comparing Davidson's branching strategy with the combination of our branching strategy and the gimmick of the redundancy check. The results include some 6-, 7-, and 8-gate functions of four variables, among others.

1. A METHOD OF REDUCING THE NUMBER OF REDUNDANT GATES AND CONNECTIONS[†] IN NON-OPTIMAL NOR NETWORKS

1.1 Elimination of a Redundant Connection

Suppose we are given NOR gates p, q, ..., t which are cascaded as shown in Fig. 1.1. Let us call this configuration a gate-chain.

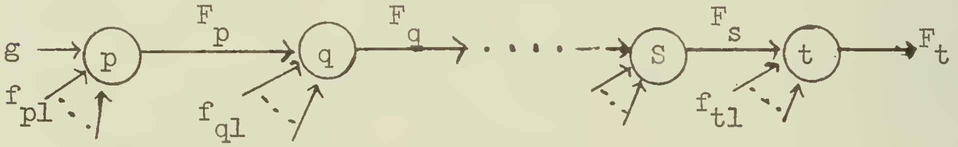


Fig. 1.1 A gate-chain consisting of gates p, q, ..., s, t.

Let g denote the input under consideration which is connected to gate p. Let f_{pi}'s be the inputs to gate p other than the input g; Let f_{qj}'s be the inputs to gate q from outside of the gate-chain; and let f_{tk}'s be the inputs to gate t from outside of the gate chain.

Property 1

In the above gate-chain, the input g is redundant with respect to the output of gate t, if and only if g implies the disjunction of the f_{pi}'s, f_{qj}'s, ..., and f_{tk}'s, i.e.,

$$(f_{p1} \vee f_{p2} \vee \ldots)\vee(f_{q1} \vee f_{q2} \vee \ldots)\vee \ldots\vee(f_{t1} \vee f_{t2} \vee \ldots) \supseteq g \quad (1.1).$$

[†] The word 'connection' denotes either an input connection from an external variable or an interconnection between two gates, unless otherwise specified.

Proof

Let Φ_p, Φ_q, ..., and Φ_t denote the disjunctions of f_{pi}'s, f_{qj}'s, ..., and f_{tk}'s, respectively. The outputs of gates in the gate-chain are given recursively as :

$$\left.\begin{array}{ll} \text{the output of gate p:} & F_p = \bar{g}\ \bar{\Phi}_p \\ \\ \text{the output of gate q:} & F_q = \bar{F}_p\ \bar{\Phi}_q \\ \vdots & \\ \text{the output of gate t:} & F_t = \bar{F}_s\ \bar{\Phi}_t \end{array}\right\} \qquad (1.2).$$

Assume that the input g is disconnected from gate p. The output of gate p changes from $\bar{g}\ \bar{\Phi}_p$ to $\bar{\Phi}_p$. This change propagates toward gate t. Let F'_p F'_q, ..., and F'_t denote the outputs of gates in the gate-chain after the disconnection of the input g from gate p:

$$F'_p = \bar{\Phi}_p,\ F'_q = \bar{F}'_p\bar{\Phi}_q\ ,\quad \ldots,\ \text{and}\ F'_t = \bar{F}'_s\bar{\Phi}_t\ .$$

The input g is redundant with respect to the output of gate t if $F_t = F'_t$ holds. The relation $F_t = F'_t$ is equivalent to $F_t \oplus F'_t = 0$. By substituting $\bar{F}_s\ \bar{\Phi}_t$ and $\bar{F}'_s\ \bar{\Phi}_t$ for F_t and F'_t, respectively, we have $F_t \oplus F'_t = \bar{\Phi}_t\ (\bar{F}_s \oplus \bar{F}'_s\ ) = \bar{\Phi}_t\ (F_s \oplus F'_s\ )$. Continuing a similar substitution process backwards to gate p, we have $0 = F_t \oplus F'_t = \bar{\Phi}_t \cdots \bar{\Phi}_q\bar{\Phi}_p\ g$.

Therefore $\bar{\Phi}_p\ \bar{\Phi}_q\ \cdots\ \bar{\Phi}_t \cdot g = 0$, or equivalently, $\Phi_p \vee \Phi_q \vee \ldots \vee \Phi_t \supseteq g$.

Q. E. D.

Note that even if the above network consists of a single gate p, the property still holds.

Suppose an input g is connected to gate p, which in turn feeds gate t through a subnetwork σ in which there are two or more paths connecting gate p to gate t (Fig. 1.2).

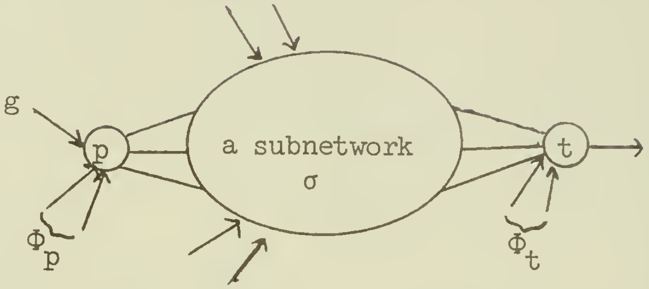


Fig. 1.2 An output g is connected to gate p, which in turn feeds gate t through a subnetwork σ.

In this case no simple expression such as (1.1) is found as a necessary and sufficient condition for g to be redundant with respect to the output of gate t. Let us introduce a function h^*_{pt} [g] with respect to g and the pair of gates p and t:

$$h^*_{pt}\ [g] = F_t \oplus F'_t \qquad (1.3),$$

where F_t is the output of gate t before disconnecting g from gate p, and F'_t is the output of gate t after diconnecting g from gate p. Clearly h^*_{pt} [g] = 0 is a necessary and sufficient condition for g to be redundant with respect to the output of gate t. Notice that in the case of h^*_{pt} [g] $\neq$ 0, the true vectors of h^*_{pt} [g] are the input vectors for which the output of gate t changes when g is disconnected from gate p.

For later use, let us derive the following two sufficient conditions for g to be redundant in Fig. 1.2 by using Property 1. The proof is omitted since it is easy to see from the proof of Property 1.

Property 2

Take a gate-chain among paths from gate p to gate t in Fig. 1.2. Let the labels p, q, ..., t denote the gates on this gate-chain (Fig. 1.3).

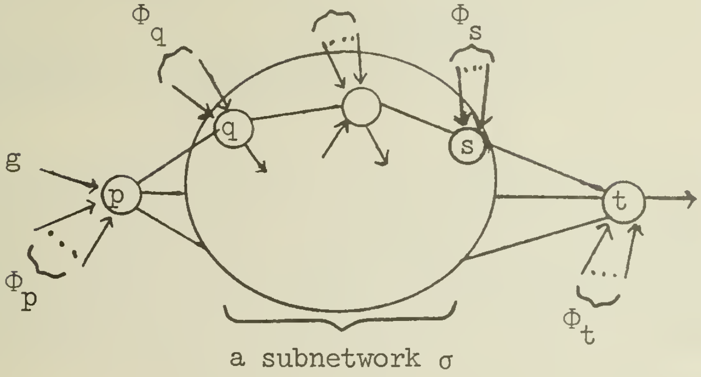


Fig. 1.3 A gate-chain from gate p to gate t in Fig. 1.2.

When $\Phi_p \vee \Phi_q \vee \dots \vee \Phi_t \supseteq g$ holds, we say the input g is redundant with respect to the output of gate t along this gate-chain, where Φ_p, Φ_q, ..., and Φ_t are the disjunctions of inputs originating outside of σ to gate p, gate q, ..., and gate t, respectively. The input g is redundant with respect to the output of gate t if g is redundant (with respect to the output of gate t) along every gate-chain connecting gate p to gate t.

Property 3

As a special case of Property 2 (Property 3 is mentioned later more often than Property 2), if $\Phi_p \vee \Phi_t \supseteq g$ holds, then g is redundant with respect to gate t (Fig. 1.4).

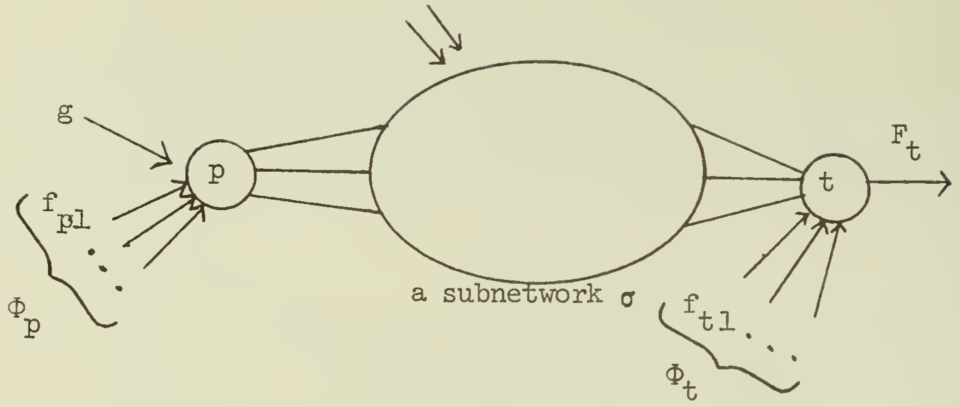


Fig. 1.4 If $\Phi_p \vee \Phi_t \supseteq g$ holds, then g is redundant with respect to gate t, where Φ_p is the disjunction of inputs (excluding g) to gate p, and Φ_t is the disjunction of inputs to gate t originating outside of the subnetwork σ.

Let us show two examples* of redundant connections which can be detected in a given network due to Property 3.

Example 1 (Triangular interconnection)

Suppose we are given a network in which gate i is connected to two gates p and t, and gate p has only one output connection which goes to gate t (Fig. 1.5).

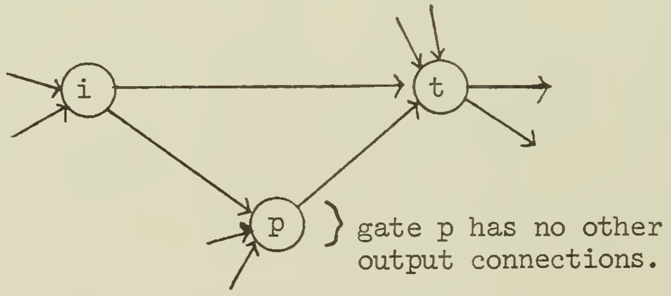


Fig. 1.5 A triangular interconnection.

* These are known as a triangular interconnection and a generalized triangular interconnection in [8].

In this configuration, the connection from gate i to gate p is redundant. This is a special case of Property 3 where the subnetwork between gates p and t consists of only a single connection. Notice that a similar property holds when gate i is replaced by an external input x_ℓ.

Example 2 (Generalized triangular interconnection)

Suppose we are given a network in which gate i is connected to a certain subnetwork σ, and to one other gate, gate t. Furthermore, all output connections from the subnetwork σ go to gate t, but not to other gates (Fig. 1.6).

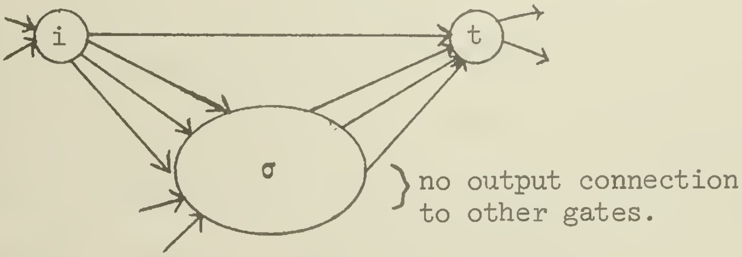


Fig. 1.6 A generalized triangular interconnection.

In this configuration, all input connections from gate i to the subnetwork σ are redundant.

1.2 Elimination Of A Gate By Substituting For Its Output Connections A Disjunction of Other Gate Outputs And/Or External Variables

Suppose we are given a network in which the outputs of two gates are identical. Obviously, if there was no fan-out restriction, we could always substitute for the output connections of one gate the output connections of the other gate and eliminate one gate and all its input connections (Fig. 1.7).

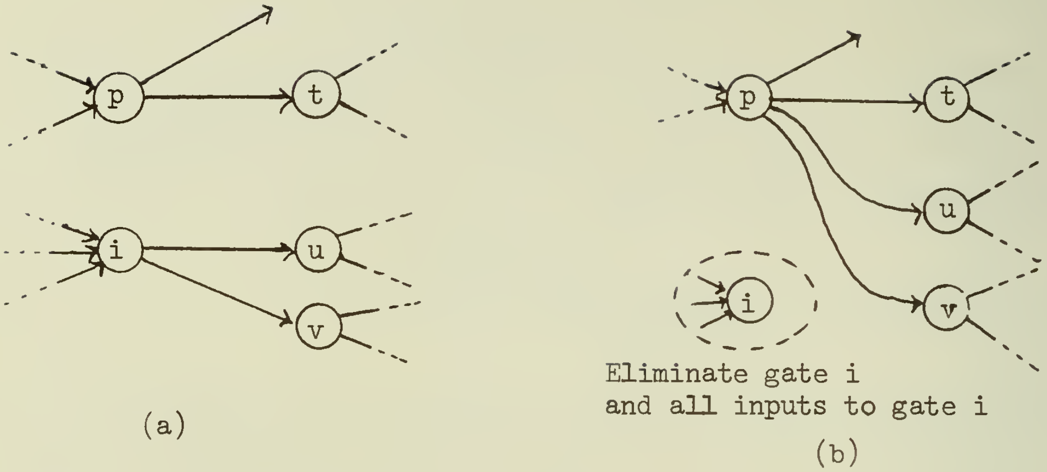


Fig. 1.7 If the output of gate i is identical to the output of gate p in (a), then a better network is possible as shown in (b).

Similarly, if a disjunction of certain external variables and/or outputs of other gates is found to be identical to the output of a gate, then we can substitute for the output connections of this gate the sum of those external variables and/or outputs of gates, eliminating one gate (Fig. 1.8). (The total number of connections could increase in this case.)

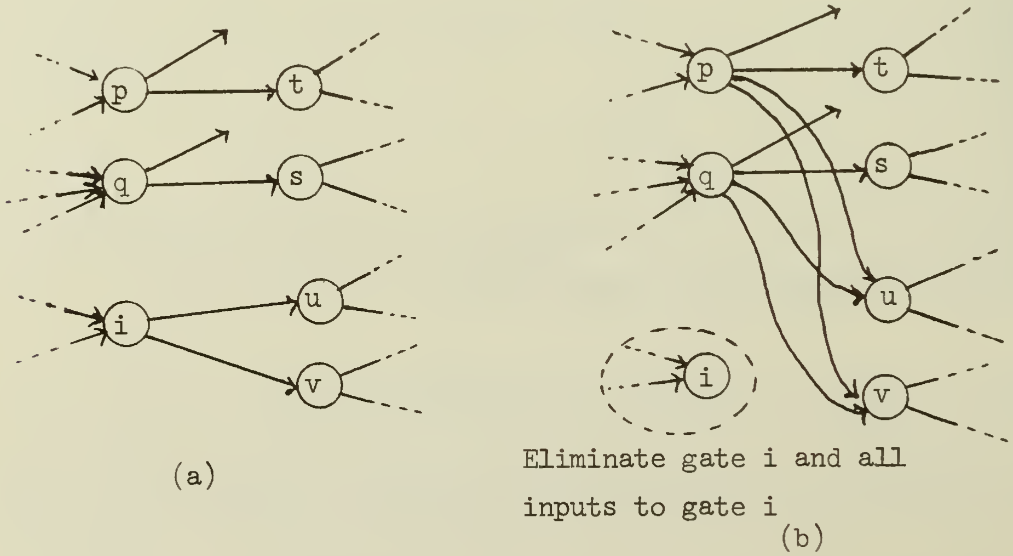


Fig. 1.8 If the output of gate i is identical to the disjunction of the outputs of gate p and gate q in (a), then we can obtain an equivalent network having one less gate as shown in (b).

1.3 Elimination Of A Gate Having A Single Output Connection By Introducing New Connections

Suppose gate i has only one output which is connected to gate p as input g. Let Φ_p denote the disjunction of inputs to gate p, excluding g. Assume that $\Phi_p \supseteq g$ does not hold. (If $\Phi_p \supseteq g$ holds, we can eliminate the input g as explained in 1.1.)

Property 4

If we have certain external variables and/or outputs of gates which are not fed by gate p such that their disjunction e satisfies the equality $\overline{\Phi}_p\,(g \oplus e) = 0$, then we can eliminate the input g from gate p after connecting these external variables and/or gate outputs directly to gate p, (Fig. 1.9), without changing the output function of gate p. Thus one gate (gate i) and all inputs to this gate are eliminated.

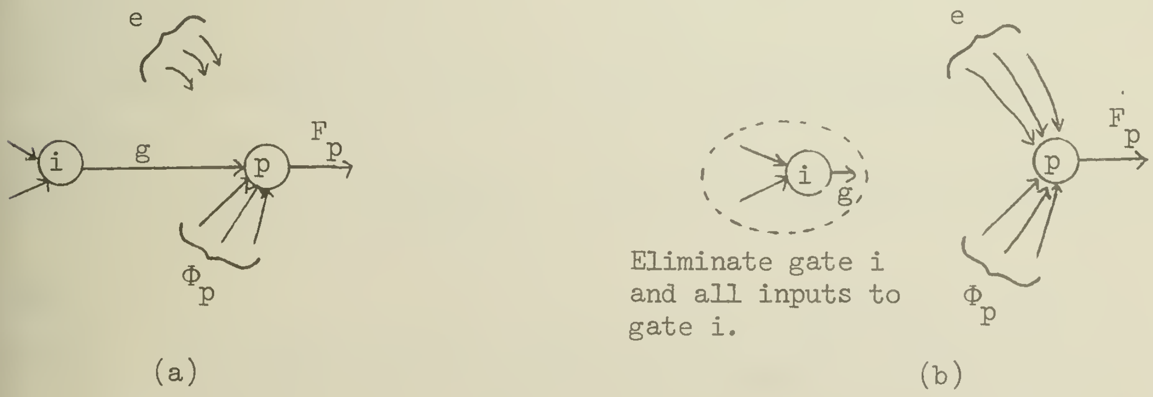


Fig. 1.9 If we have certain external variables and/or outputs of gates whose disjunction, e, satisfies $\overline{\Phi}_p\,(g \oplus e) = 0$ in (a), then we can eliminate the input g after connecting those external variables and/or gate outputs to gate i, as shown in (b).

Proof

From $\overline{\Phi}_p\,(g \oplus e) = 0$, we have $\overline{\Phi}_p\,\overline{g} \oplus \overline{\Phi}_p\,\overline{e} = 0$, which is equivalent to the equality $\overline{\Phi}_p \; \overline{g} = \overline{\Phi}_p \; \overline{e}$. Therefore a replacement of g by e does not change the output $F_p = \overline{\Phi}_p \; \overline{g}$ of gate p. Q.E.D.

We can easily extend this property as follows. Suppose we are given a network in which a gate p, a subnetwork σ_1, a gate r, a subnetwork σ_2, and a gate t are cascaded as shown in Fig. 1.10. The only output connection of gate i is connected to gate p as input g.

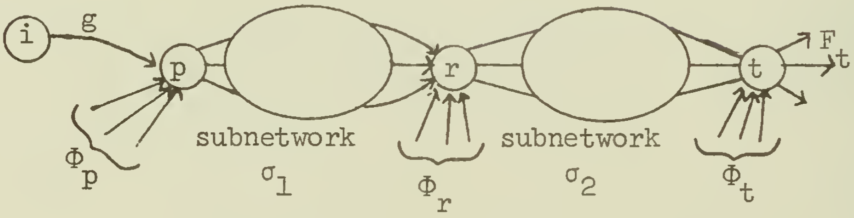


Fig. 1.10 Gate p, subnetwork σ_1, gate r, subnetwork σ_2, and gate t are cascaded, where the input g to gate p is the only output connection of gate i.

Let us define two functions h^*_{pt} and e^*_{rt} as follows:

$$\text{(i)} \quad h^*_{pt} = F_t \oplus F'_t \qquad (1.4),$$

where F_t is the output of gate t before disconnecting g from gate p, and F'_t is the output of gate t after disconnecting g from gate p. The function h^*_{pt} is the same as (1.3). (For simplicity, [g] is dropped.) Recall that the true input vectors of h^*_{pt} are the input vectors for which the output of gate t changes after disconnecting the input g from gate i. Define

$$\text{(ii)} \quad e^*_{rt} = \overline{F_t \oplus F''_t} \qquad (1.5),$$

where the F_t is the same as the one used for h^*_{pt}, and F''_t is the output of gate t obtained by connecting the constant 1 (one) to gate r.

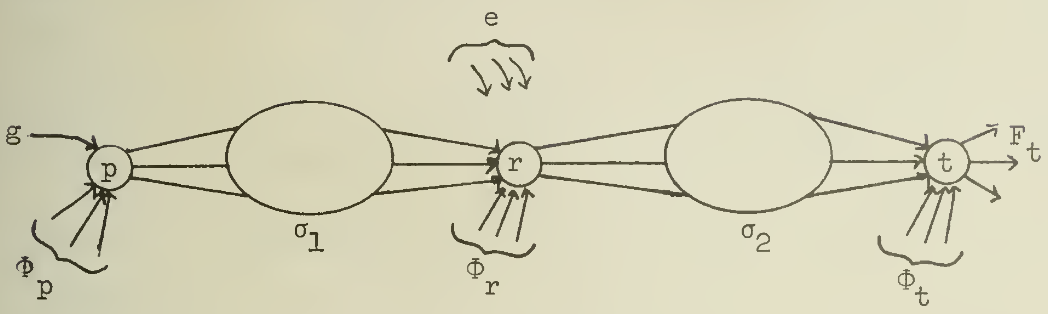


Fig. 1.11 If e satisfies $e^*_{rt} \supseteq e \supseteq h^*_{pt}$, then e can be connected to gate r without changing the output of gate t.

Property 5

In Fig. 1.11, if we have certain external variables and/or outputs of outside gates such that their disjunction e satisfies

$$h^*_{pt} \subseteq e \subseteq e^*_{rt} \qquad (1.6),$$

then, without changing the output of gate t, we can eliminate the input g from gate p after connecting these gate outputs and/or external variables (whose disjunction is e) to gate r.

Proof

Consider a true vector E of e^*_{rt}. Then by (1.5) $F_t \oplus F''_t = 0$ since $e^*_{rt}=1$ for E. In other words $F_t = F''_t$. Thus, even if e^*_{rt} or e (since $e \subseteq e^*_{rt}$) is connected to gate r, the output of gate r does not change. Therefore it suffices to prove that the disconnection of g from gate i does not change the output of gate t when e is already connected to gate r. For this purpose, let us introduce a function h^*_{pr}, similar to h^*_{pt}, defined by $h^*_{pr} = F_r \oplus F'_r$, where F_r is the output of gate r in the original network, and F'_r is the output of gate r when g is disconnected from gate p but e is not yet connected to gate r. Clearly $h^*_{pr} \supseteq h^*_{pt}$. Now, assume that e is connected to gate r. Using h^*_{pr}, let us define another function $h^{*\prime}_{pr} = h^*_{pr}\,\bar{e}$. The true input vectors of the function $h^{*\prime}_{pr}$ are the input vectors for which the output of gate r changes by disconnecting g from gate p. Thus, the true input vectors of $h^{*\prime}_{pr}$ are the only input vectors for which the output of gate t might change by disconnecting g from gate p. However, the equality $h^{*\prime}_{pr} \cdot h^*_{pt} = 0$ holds due to $h^{*\prime}_{pr} = h^*_{pr}\,\bar{e}$ and the assumption of $e \supseteq h^{*\prime}_{pt}$. In other words, if $h^{*\prime}_{pr} = 1$, then h^*_{pt} must be 0. Therefore those changes in gate r corresponding to the true input vectors of $h^{*\prime}_{pr}$ do not affect the output of gate t. Q.E.D.

Notice that e^*_{rt} is the maximal Boolean function which can be connected to gate r without changing the output of gate t (Fig. 1.11).

In the computational procedure which will be presented in 1.6, we implement the following special three cases because it would be too time-consuming to implement general Property 5. Cases A and B are special cases of Property 5, and case C is an extension of cases A and B (A proof is omitted since it is easy).

Property 5 for the following cases:

Case A: $r = p$ (Fig. 1.12 (a)),

Case B: $r = t$ (Fig. 1.12 (b)), and

Case C: a combination of cases A and B, assuming that gate(s) t are the output gate(s) of the entire network.

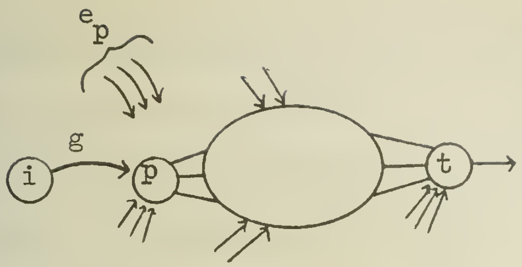


Fig. 1.12 (a)

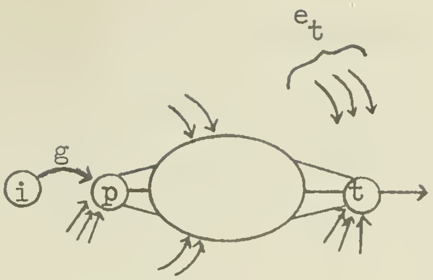


Fig. 1.12 (b)

Case A: e_p can replace g, if e_p satisfies $h^*_{pt} \subseteq e_p \subseteq e^*_{pt}$.

Case B: e_t to gate t can replace g, if e_t satisfies $h^*_{pt} \subseteq e_t \subseteq e^*_{tt}$.

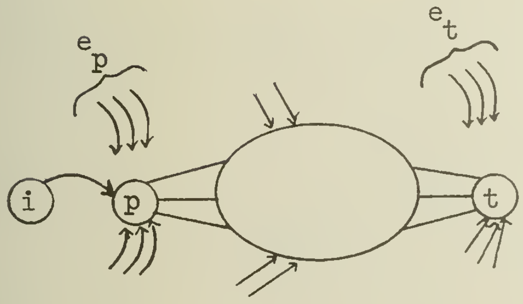


Fig. 1.12 (c)

Case C: e_p and e_t together replace g, if e_p and e_t satisfy $e_p \vee e_t \supseteq h^*_{pt}$, $e_p \subseteq e^*_{pt}$, and $e_t \subseteq e^*_{tt}$ simultaneously.

It is difficult, if not impossible, to extend Property 5 to the general case where some gate-chains from gate p to gate t do not contain gate r (Fig. 1.13).

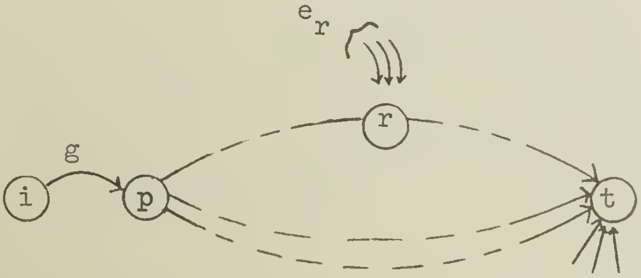


Fig. 1.13 Some gate-chains from gate p to gate r do not go through gate r.

Alternatively, let us investigate Fig. 1.14 as a special case of Fig. 1.13. In Fig 1.14, we attempt to eliminate g by connecting certain gate outputs and/or external variables to each gate immediately preceding the output gate(s) of the entire network. We call this case Case D . Let gate r_k, k=1, ..., α denote such gates immediately preceding the output gate t which are fed by gate p directly or through other gates. Let F_{r_k} denote the output of gate r_k, k=1, ..., α, and let Φ_t denote the disjunction of inputs to gate t from outside of the subnetwork shown.

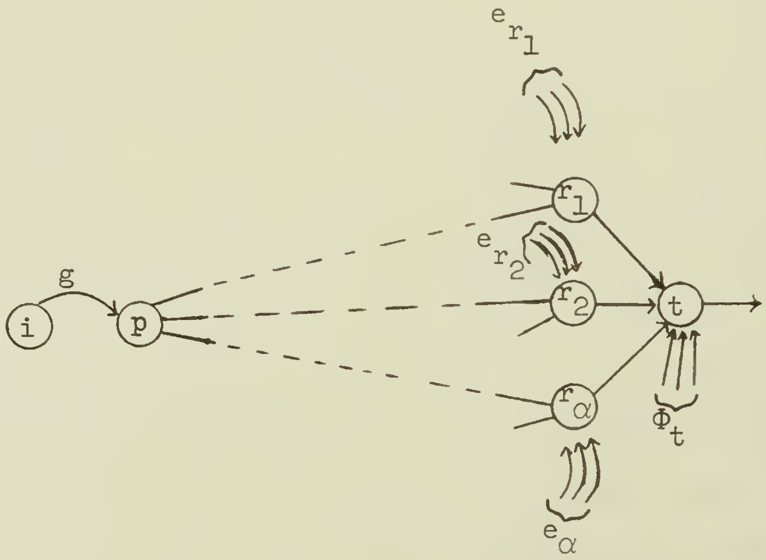


Fig. 1.14 Case D: We attempt to eliminate g by connecting e_{r_k} to gate r_k for k=1, ..., α, respectively.

Property 5'

If we have α sets of certain gate outputs which are not fed by gate p directly or through other gates, and/or external variables whose disjunctions e_{r_k} , k = 1, ..., α, satisfy the following conditions simultaneously;

$$\left.\begin{aligned} h^*_{pt} &\subseteq e_{r_1} \subseteq \overline{F}_{r_1} \vee \Phi_t \\ h^*_{pt} &\subseteq e_{r_2} \subseteq \overline{F}_{r_2} \vee \Phi_t \\ h^*_{pt} &\subseteq e_{r_\alpha} \subseteq \overline{F}_{r_\alpha} \vee \Phi_t \end{aligned}\right\} \qquad (1.7),$$

then we can eliminate g after connecting e_{r_k} to gate r_k for all k=1, ..., α, respectively.

Proof

Consider an arbitrary vector a of the function $\overline{F}_{r_k} \vee \Phi_t$. Then Φ_t=1 or $\overline{F}_{r_k}$ =1 holds for a. Thus, even if a is connected to gate r_k, the output of gate t does not change since Φ_t=1 or F_{r_k} =0 (Note that gate r_k is a NOR gate). Therefore, even if e_{r_k} such that $e_{r_k} \subseteq \overline{F}_{r_k} \vee \Phi_t$ is connected to gate r_k, the output of gate t does not change. Now assume we connect e_{r_k} to gate r_k, for all k=1, ..., α. Since e_{r_k} satisfies $h^*_{pt} \subseteq e_{r_k}$ for all k=1, ..., α, the disconnection of g from gate p does not affect the output of gate t, as is shown in the proof of Property 5. Q.E.D.

In the computational procedure, we check Case D, when none of the cases A, B, and C lead to the disconnection of g. Furthermore, if case D does not lead to the disconnection of g, we attempt to eliminate g by connecting certain gate outputs (which are not fed by gate p directly or through other gates) and/or external variables to all of gates p, t, and gate r_k, for all k=1, ..., α, as a combination of Case A, Case B and Case D. We call this case Case E.

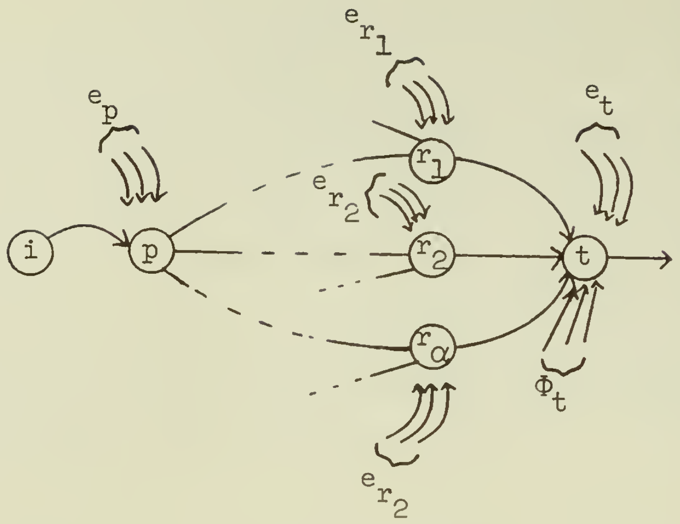


Fig. 1.15 Case E: We attempt to eliminate g by connecting e_p, e_t, e_{r_1}, ..., e_{r_α}, simultaneously.

1.4 Elimination Of Connections By Introducing A New Connection

Suppose an input g is connected to a subnetwork σ, which in turn feeds gate t, as shown in Fig. 1.16 (a).

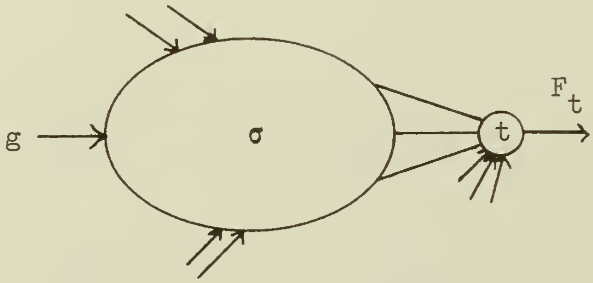


Fig. 1.16 (a) An input g is connected to a subnetwork σ whose output connections go to gate t.

Property 6

In Fig. 1.16 (a), if g implies $\overline{F}_t$, the negation of the output F_t of gate t, then the new network configuration of Fig. 1.16 (c) is equivalent to Fig. 1.16 (a) with respect to the output of gate t.

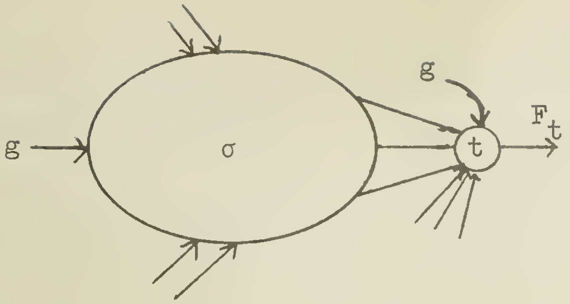


Fig. 1.16 (b) An intermediate stage of the reconfiguration of Fig. 1.16 (a). A duplicate g is added to gate t.

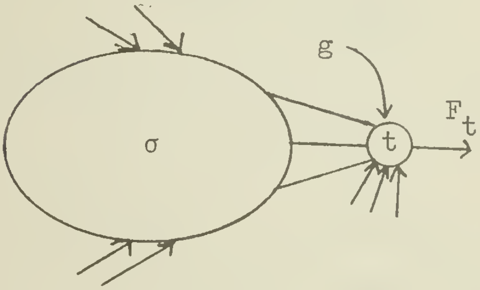
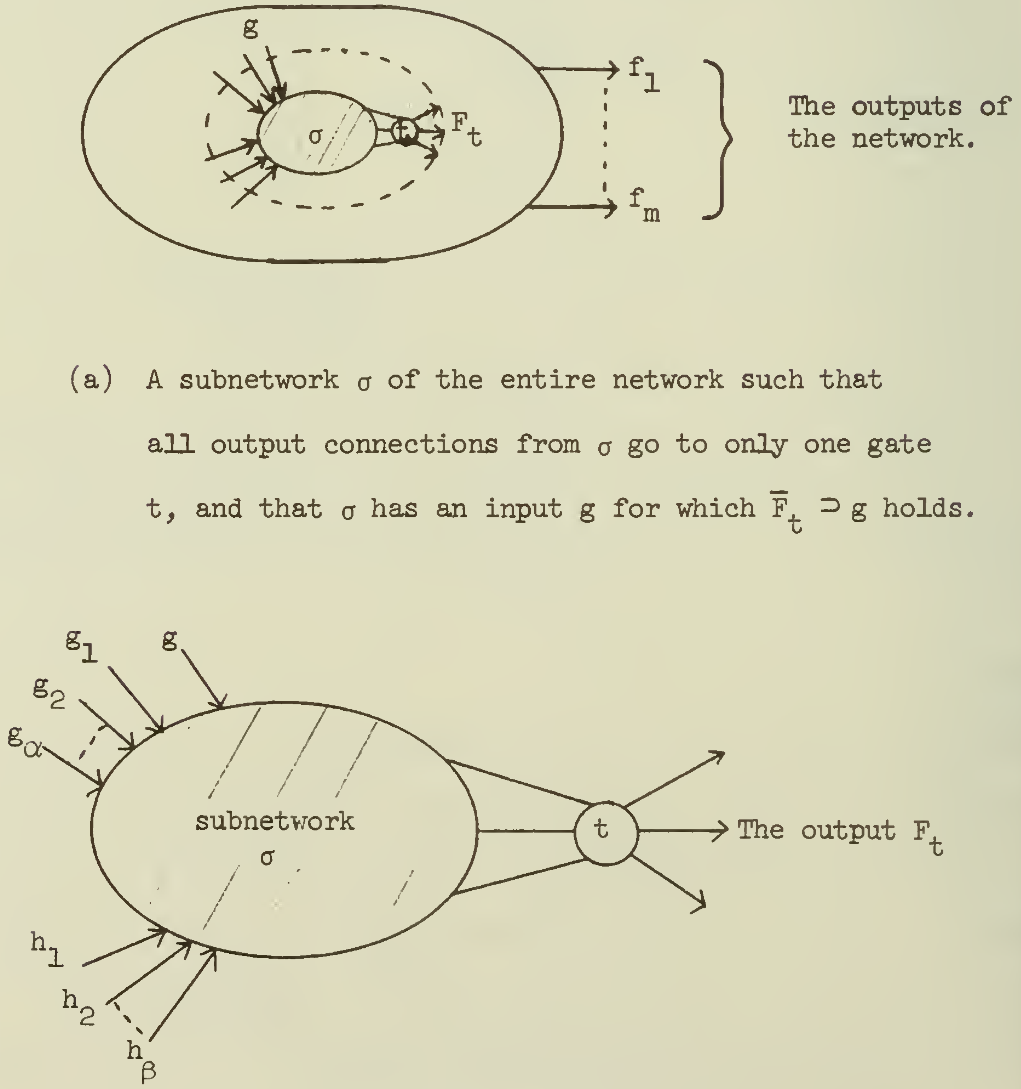


Fig. 1.16 (c) A duplicate g is connected to gate t, and the original g is disconnected from the subnetwork σ.

Proof

Consider an arbitrary true vector a of the function of $\overline{F}_t$. Then $F_t=0$ for this a. This means that at least one input of NOR gate t assumes the value 1. Therefore even if g is duplicated at the input of gate t (see Fig. 1.16 (b)), the output of gate t does not change if $g \subseteq \overline{F}_t$. Uning Property 3, we know the input g which is connected to the subnetwork σ is redundant with respect to gate t. Q.E.D.

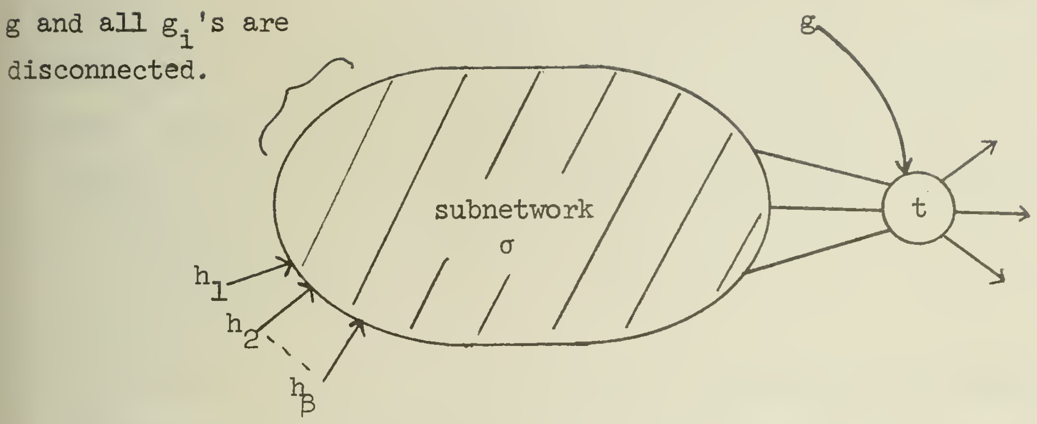
In the computational procedure which will be presented in Section 1.6, we implement Property 6 as follows. Consider a subnetwork σ of the entire network such that all output connections from σ go to only one gate. Let gate t denote this gate. Assume that σ has an input g for which $\overline{F}_t \supseteq g$ holds, where F_t is the output of gate t (Fig. 1.17 (a)). Furthermore, let g_i, $i=1, \ldots, \alpha$, denote the inputs to σ, if any, such that each g_i implies g, i.e., $g \supseteq g_i$. Let h_j, $j=1, \ldots, \beta$ denote the inputs to σ other than g_i (Fig. 1.17 (a)).



(a) A subnetwork σ of the entire network such that all output connections from σ go to only one gate t, and that σ has an input g for which $\overline{F}_t \supseteq g$ holds.

(b) The part of the entire network under consideration where $\overline{F}_t \supseteq g$, and $g \supseteq g_i$ for $i=1, \ldots, \alpha$ hold.

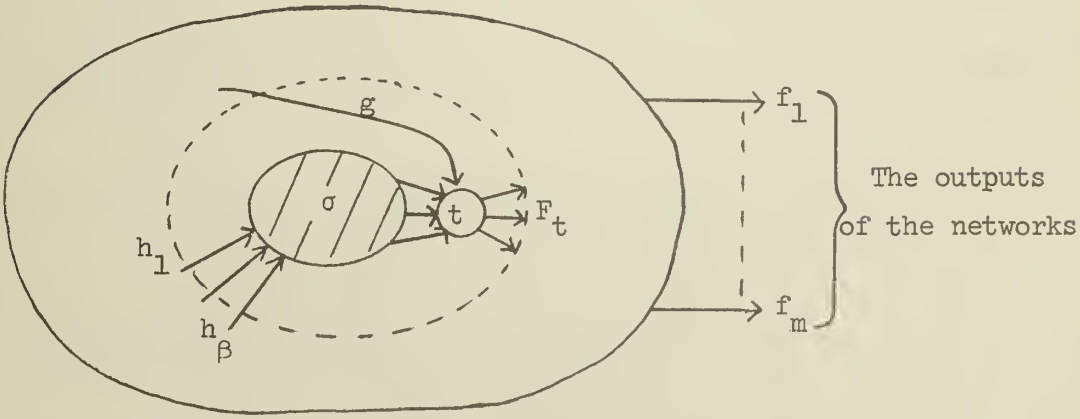
Fig. 1.17 Eliminating connections using Property 6.



(c) This configuration is derived from (b) by

(i) connecting a duplicate of g to gate t and

(ii) disconnecting g and all g_i's from σ.



(d) The entire network derived by replacing (b) by (c).

Fig. 1.17 (cont'd) Eliminating connections using Property 6.

In the above configuration (b), first connect a duplicate of g to gate t, and then disconnect g from σ. Since $g \supseteq g_i$ holds for i=1, ..., α, we disconnect all g_i's from σ, by using Property 3. Fig. 1.17 (c) shows this transformation, and Fig. 1.17 (d) is the resulting entire network. Clearly Fig. 1.17 (d) has α-1 fewer connections than the original network of Fig. 1.17 (a). Notice that when σ has no g_i's for which $g \supseteq g_i$ holds, this transformation does not yield a reduction in the number of connections. But we will derive a new network of Fig. 1.17 (d) of the same number of connections, even in this case, because the new network has a different configuration, so that it can be used for further redundancy checking.

1.5* A Transformation Which May Allow The Elimination Of Some Connections

Assume we have a network configuration as shown in Fig. 1.18 (a), where

(i) the outputs of the subnetwork σ go to two different gates r and t,

(ii) the output of gate r goes only to the third gate s,

(iii) the output of gate s goes only to gate t,

(iv) gate s has one or more inputs g_1, g_2, ..., g_α other than the input from gate r, and

(v) the disjunction of all g_i's implies the output F_t of gate t, i.e., $F_t \supseteq g_1 \vee g_2 \vee \dots \vee g_\alpha$.

[*] We do not implement this checking scheme in our current computational procedure which will be presented in Section 1.6, because this case appears to occur only rarely judging from our preliminary experiments with the branch-and-bound algorithm.

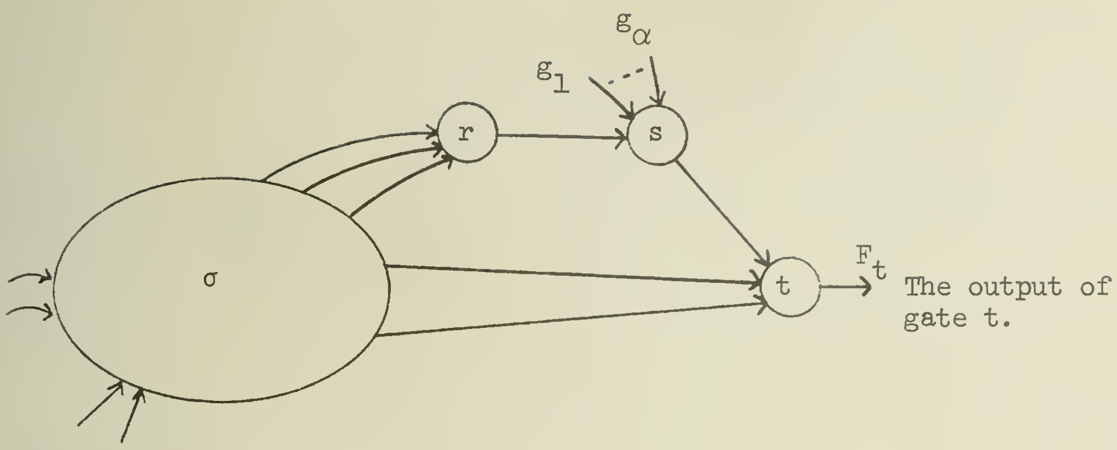


Fig. 1.18 (a) A network configuration which satisfies the assumptions (i) ~ (v).

Property 7

Given Fig. 1.18 (a), we can obtain another network configuration Fig. 1.18 (c) which is equivalent to Fig. 1.18 (a) with respect to the output of gate t.

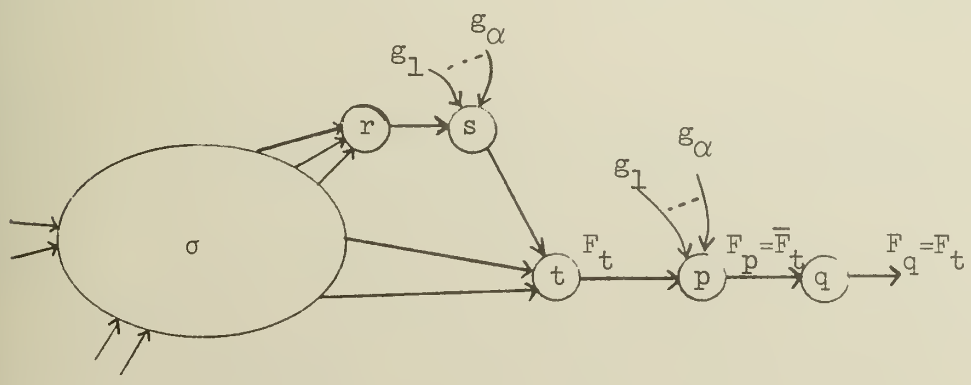


Fig. 1.18 (b) An intermediate stage of the reconfiguration of Fig. 1.18 (a)

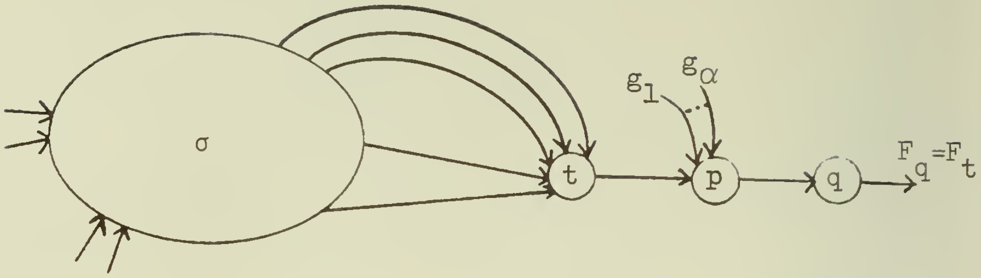


Fig. 1.18 (c) A network configuration equivalent to Fig. 1.18 (a), where

(i) gate r and gate s are eliminated,

(ii) the output connections from σ to gate r are connected to gate t ,

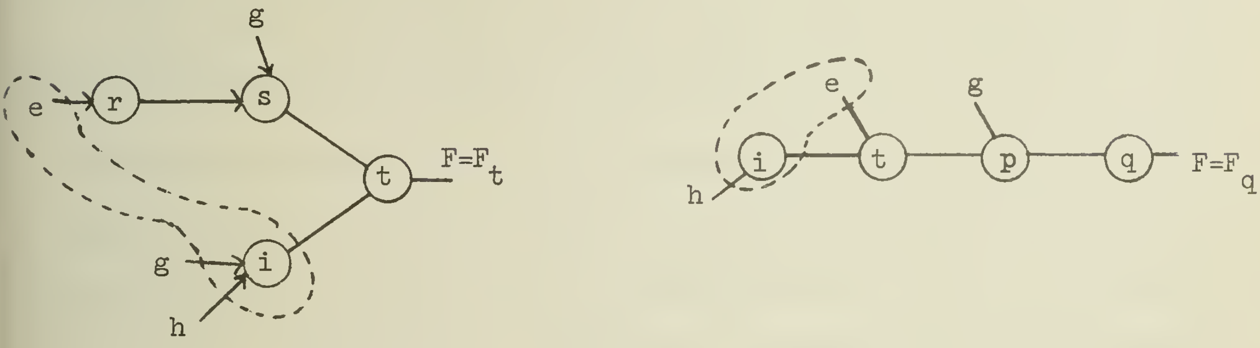
(iii) two new gates p and q are connected in series following gate r, and

(iv) $g_1, \ldots, g_\alpha$ are connected to gate p.

Proof

First, let us make a double negation of the output of gate t by connecting two single input NOR gates p and q in series after gate t. Since the output, F_p, of gate p is identical to $\overline{F}_t$, $\overline{F}_p$ is implied by $g_1 \vee \ldots \vee g_\alpha$ due to the assumption (v). Therefore we can connect $g_1, \ldots, g_\alpha$ to gate p. This results in Fig. 1.18 (b). Using Property 3 with respect to gate p, we can eliminate all g_i's which are connected to gate s. After an elimination of all g_i's from gate s, gate s has only one input connection and one output connection. Therefore we can eliminate gate r and gate s, resulting in Fig. 1.18 (c). Q.E.D.

Fig. 1.19* shows a simple example of the transformation of Fig. 1.18. (After applying to Fig. 1.19 (a) the transformation of Fig. 1.18, we eliminate the input g to gate i by using Property 3, obtaining Fig. 1.19 (b)).



(a) The original network realizing F. The dotted line indicates the subnetwork σ for this case.

(b) A new network realizing the same F. The network has one less connection than the original network (a).

Fig. 1.19 An example of reconfiguration explained in Fig. 1.18.

* This reconfiguration was first found by J. Culliney. We generalized his finding later, resulting in Property 7.

1.6 A Computational Procedure To Attempt To Improve Redundant Networks By Transformations

Using the network transformations explained in Sections 1.1 through 1.4, we implement a computational procedure to generate an improved network from a given (non-optimal*) network. For the implementation of a practical computational procedure, we have to compromise between the complexity and the power of the procedure. (An excessively complicated procedure may increase the total computation time of the algorithm, even while lowering the cost ceiling substantially by finding a much improved network.) Thus, instead of incorporating all of the ideas in the previous sections, only simple cases are incorporated. The principle of the implementation is:

(i) to attempt to eliminate redundant gates first,

(ii) then to attempt to eliminate redundant connections.

Let us first present the segments of the computational procedure which we implemented.

Part I (Replacement of the output of a gate with a disjunction of other gate outputs and/or external variables.):

Select a gate p which is not an output gate of the network. Let gates t', t", ..., denote the gates to which gate p is connected. If there are certain external variables and/or outputs of gates which are not fed (directly or through other gates) by gate p such that their disjunction is

* When a given network is optimal, some transformations may yield other optimal networks.

identical to the output of gate p, then make new connections from these external variables and/or gates to gates t', t", ..., and remove the existing connections from gate p to gates t', t", ..., along with all input connections of gate p. By this transformation, we eliminate one gate, i.e., gate p (although the number of connections might be increased).

Part II (Elimination of a gate having a single output connection)

Select a gate i which is not an output gate, and which has only one output connection. Let gate p denote the gate to which gate i is connected. Check whether or not $h^*_{pt} = 0$ (h^*_{pt} is defined by (1.4)) for all gates t which are the output gates of the network. If the connection is redundant, (i.e., $h^*_{pt} = 0$) then remove the connection. By this transformation, we eliminate gate i and its inputs and output. If the connection is not redundant (i.e, $h_{pt} \neq 0$ for some t), then by applying Properties 5 or 5' we determine if there are any external variables and/or outputs of gates which are not fed by gate p, such that by introducing new connections from these external variables and/or gates, the connection of gate i to gate p becomes redundant.

As we explained in Section 1.3, the detailed checking procedure is: First check Case A (Fig. 1.12 (a)). If Case A does not eliminate the gate having a single output connection, then check Case B (Fig. 1.12 (b)), Case C (Fig. 1.12 (c)), Case D (Fig. 1.14), and Case E (Fig. 1.15), successively.

Part III-a (Elimination of a redundant connection of a gate)

Select a gate i, which is not an output gate and has two or more output connections, or which is an output gate having output connections.

Determine whether or not an output connection from gate i to another gate, p, is redundant with respect to the output(s) of the entire network; this is done by checking whether or not $h^{*}_{pt}=0$ (h^{*}_{pt} [g] is defined by (1.3)), where g is the output connection under consideration from gate i to gate p, and t denotes the output gate(s) of the entire network. If h^{*}_{pt} [g] = 0 for all output gates t, then remove the connection g.

Part III-b (Elimination of a redundant connection of an external variable)

Select an external variable x_{ι}. Determine whether or not the connection of x_{ι} to a certain gate is redundant with respect to the output(s) of the entire network; this is done by checking whether or not h^{*}_{pt} [g] = 0, where g is the connection (of x_{ι}) under consideration to gate p, and t denotes the output gate(s) of the entire network. If h^{*}_{pt} [g] = 0 for all output gates t, then eliminate the connection g.

Part IV (Elimination of redundant connections by the introduction of a new connection)

Select a portion of the entire network which consists of gate t and a subnetwork σ, where the outputs from σ go only to gate t (Fig. 1.17(a)). When an input g to σ satisfies $\bar{F}_t \supseteq g$, where F_t is the output of gate t, derive a new network as shown in Fig. 1.17. By this reconfiguration we eliminate α-1 connections. (As noted, α-1 could be 0.)

Combining Parts I through IV, we complete the computational procedure for generating an improved network from the given (non-optimal) network.

2. THE BRANCHING STRATEGY[†]

2.1 Definitions

Before presenting the branching strategy, we present the necessary definitions. The problem is to find NOR networks having minimal cost for given switching functions, where the cost C of a network is defined by A × R + B × I in which R is the number of gates, I is the number of inputs to gates, and A and B are arbitrary non-negative weights. Suppose the problem is to obtain the optimal networks each of which realizes the given m output functions f_i, i=1, ..., m, of n variables, simultaneously. The initial solution is defined by a set of m gates; each gate i is assigned the output function f_i, i=1, ..., m.

We represent the output of a gate with a 2^n-tuple of 0-1 variables P^j, as $(P^0, \ldots, P^{2^n-1})$, where P^j corresponds to the j-th input vector for j=0, ..., 2^{n-1}. Notice, however, that the algorithm will not immediately assign a definite value 0 or 1 to all the P^j's of a gate, so a symbol * is used to denote an unassigned value of P^j. The output of gate i, i=1, ..., m, in the initial solution is $(f_i^0, \ldots, f_i^{2^n-1})$.

Let us consider two gates, i and k, in a NOR network, where gate i is connected to gate k. Let $(P_i^0, \ldots, P_i^{2^n-1})$ and $(P_k^0, \ldots, P_k^{2^n-1})$ denote the outputs of gate i and gate k, respectively. Because the gates perform a NOR operation, the outputs of the gates must satisfy the following condition;

[†] The concepts used for making our branching strategy are Davidson's [1, 2]. The details of this branching strategy are discussed in [3].

$$\left.\begin{array}{l} P_k^j = 0 \text{ for all } j \text{ such that } P_i^j = 1 \\ \\ P_i^j = 0 \text{ for all } j \text{ such that } P_k^j = 1 \text{ .} \end{array}\right\}$$

A similar condition must hold for the output of each gate in a NOR network with respect to each of the connected external variables, x_ℓ.

When the output of a gate in a NOR network satisfies the above condition with respect to all immediately preceding/succeeding gates and all connected variables, we call this output <u>a feasible assignment</u>. There can be unassigned components in a feasible assignment.

<u>An intermediate solution</u> is defined as a network of R gates, $R \geq m$, where the outputs of the first m gates are identical to those assigned in the initial solution; the outputs of other gates (gate i for i=m+1, ..., R), if any, are completely or incompletely specified; and the output of each gate i, i=1, ..., R, is a feasible assignment.

Consider a 0-component, $P_k^{j_0}$, in gate k of an intermediate solution. If gate k has at least one input (from an external variable or another gate) whose j_0-th component is 1, then $P_k^{j_0}$ is said to be <u>covered</u>. Otherwise, $P_k^{j_0}$ is said to be <u>uncovered</u>.

Our algorithm to derive an optimal network generates from the initial solution an intermediate solution in which all the 0-components of all gates are covered. To facilitate our discussion, we introduce the concept of possible covers for an uncovered component:

<u>A possible cover</u> for an uncovered component $P_k^{j_0} = 0$ in gate k is an external variable or a gate which satisfies one of the following conditions:

(i) An external variable, x_ℓ , which is not yet connected to gate k, and which has $x_\ell^{j_0} = 1$, and $x_\ell^j = 0$ for all j such that $P_k^j = 1$. (2.1)

(ii) A gate i which is connected to gate k, and $P_i^{j_0} = *$ (unassigned). (2.2)

(iii) A gate i which is not yet connected to gate k, whose connection to gate k will not form any loops, which has $P_i^{j_0} = 1$ or $*$, and $P_i^j = 0$ or $*$ for all j such that $P_k^j = 1$. (2.3)

(iv) A gate which is not yet incorporated into the intermediate solution. (This is called a new gate.) The output components of this gate are all $*$ (unassigned), and this gate is numbered R+1, when the given intermediate solution has R gates. (2.4)

Consider a possible cover for an uncovered component $P_k^{j_0} = 0$ in gate k of the given intermediate solution S. We cover $P_k^{j_0}$ with the possible cover by the following implementation consisting of three steps: (Step 1) If the possible cover is not yet connected to gate k, then connect it to gate k. (Step 2) If the j_0-th component of the possible cover is not yet assigned, then assign the value 1 to it. (Step 3) Assign the value 0 to unassigned components wherever such an assignment is necessary to make each output assignment feasible. The resulting intermediate solution is called the augmented intermediate solution of the given intermediate solution S.

Starting from the initial solution, if we repeatedly cover uncovered components in intermediate solutions, we may eventually obtain an intermediate solution in which the 0-components in all of the gates are covered.

We call such an intermediate solution a feasible solution. Optimal solutions are feasible solutions having the least cost among all feasible solutions of a given problem. To enumerate all feasible solutions in a systematic way, we introduce a set of two rules, namely, the selection criterion of uncovered components (SCUC) and the implementation priority of possible covers (IPPC) as defined below.

The SCUC is the criterion under which an uncovered component, $P_k^j=0$, is selected to be covered from an intermediate solution.

The IPPC is the criterion under which the order of implementation is determined among the possible covers of a selected uncovered component.

We call the heuristics of the SCUC and the IPPC the branching strategy. In order to obtain a 'good' branching strategy, let us introduce more definitions in the following. Examine a component $P_k^{j_0} = 0$ of gate k. If $P_k^{j_0} = 0$ is covered, then $P_k^{j_0} = 0$ is said to be of type COV. For an uncovered component, $P_k^{j_0} = 0$, of gate k, types of possible covers are defined as follows.

Gate i which satisfies (2.2) is said to be of type G*.

External variable x_ℓ which satisfies (2.1) is said to be of type VC*.

Gate i which satisfies (2.3) is assigned one of the following four types according to the values of its components:

Type GC*0, if $P_i^{j_0} = 1$, and $P_i^j = 0$ for all j such that $P_k^j = 1$; Type GC*0*, if $P_i^{j_0}$ = 1, and there is at least one j such that $P_i^j = *$ for $P_k^j = 1$; Type G*C*0, if $P_i^{j_0} = *$, and $P_i^j = 0$ for all j such that $P_k^j = 1$; Type G*C*0 if $P_i^{j_0} = *$, and there is at least one j such that $P_i^j = *$ for $P_k^j = 1$.

A new gate of definition (2.4) is said to be of Type NWG.

Among the types introduced above, we define a desirability order, and then we defi**n**e the types of uncovered components, and the types of gates, by means of the desirability order.

The desirability order of types is the order defined by

COV - G* - VC* - GC*O - GC*O* - G*C*O - G*C*O* - NWG.

(COV is the most desired)

The type of an uncovered component is defined to be the most desirable type among the types of its possible covers.

The type of a gate is defined as the least desirable type among all the O-components in the gate.

2.2 Branching Strategy

Based on the concepts defined in Section 2.1, the branching strategy, i.e., the selection criterion of uncovered components (SCUC) and the implementation priority of possible covers (IPPC), is as follows.

If the current intermediate solution contains type NWG components, we employ Davidson's special scheme called 'Remove NF Vectors'*, which generates from the original intermediate solution with type NWG components another intermediate solution in which no type NWG components exist. This scheme is to choose certain type NWG components to be covered in order to maximize the number of new gates which must be introduced into the network. This results in the preclusion of some equivalent networks with permuted gate labels. For details, refer to [2]. Therefore, we

* This is explained in [2], as 'Special Treatment of Type NWG Components.'

assume in the following that the current intermediate solution does not have uncovered components of type NWG.

The SCUC

If the current intermediate solution has uncovered components of types VC* or less desirable, then select an uncovered component with the fewest possible covers, among those whose types are VC* or less desirable. If there are two or more uncovered components which have the same least number of possible covers, then select one whose type is the least desirable.

If the current intermediate solution has only uncovered components of type G*, then select an uncovered component (of type G*) which has the fewest possible covers.

The IPPC

Implement the possible covers of the selected uncovered component according to the following order

G* - VC* - {GC*O, GC*O*, G*C*O, G*C*O*}[†]- NWG.

(The possible cover(s) of type G* are assigned the highest priority.)

The motivation of establishing this set of the SCUC and IPPC is as follows:

An uncovered component which has a smaller number of possible covers is 'hard to cover' in a sense, since if we postpone a covering of this uncovered component until later, this uncovered component will most likely lose all possible covers except a new gate. Accordingly, we might miss

[+] The possible covers of types GC*O, GC*O*, G*C*O, and G*C*O* are assigned an equal priority.

good networks in the early stages of the computation. Therefore, it seems a good rule to first cover an uncovered component which has the fewest possible covers in a given intermediate solution. However, if we take simply "the fewest possible covers" as the main criterion of selecting uncovered components without distinguishing type G* components from uncovered components, type G* components could be selected too frequently at an early stage of the branching steps. Concentration on covering type G* components may likely result in the following situation: one gate which is introduced as a new gate to cover a certain component of another gate realizes the negation of the latter gate. With the repetition of this process, we may create an indefinite number of inverters in series without realizing a feasible network. In order to avoid such a hazard, we select one uncovered component with the fewest possible covers among those whose types are VC* or less desirable. When the given intermediate solution has only uncovered components of type G*, we select one with the fewest possible covers.

Once an uncovered component is selected, we cover this uncovered component with each of its possible covers, generating the corresponding augmented intermediate solutions. We have to order these possible covers in order to generate 'good' augmented intermediate solution first, i.e., one which is as close to a feasible solution as possible. The desirability order defined in Section 2.1 seems to be a good rule for this purpose.

2.3 Differences of Our Branching Strategy From Davidson's [†]

The branching strategy presented in Section 2.2 is different from Davidson's in the following three aspects.

(i) Uncovered components of type 1-COV.

Davidson classifies uncovered components which have type G* possible covers into two categories, namely, uncovered components of type 1-COV, and uncovered components of type G*. An uncovered component $P_k^{j_0}$ of gate k is assigned type 1-COV (a) if $P_k^{j_0}$ has at least one possible cover gate i of type G*, and (b) if the j_0-th component of every gate immediately preceding gate i is 0. An uncovered component which has type G* possible covers none of which satisfy condition (b) above is assigned type G*. Type 1-COV $P_k^{j_0}$ has the important property that covering of $P_k^{j_0}$ with such a gate i does not introduce any uncovered components into the augmented intermediate solution (Fig. 2.1). Accordingly, when an intermediate solution contains only uncovered components whose types are 1-COV, this intermediate solution has already realized the given output function(s). Because of this property of type 1-COV, type 1-COV uncovered components need not be covered explicitly. For this reason, the SCUC is applied only to the uncovered components whose types are not 1-COV, in Davidson's branching strategy.

[†] Davidson's strategy discussed in this paragraph is the version claimed by him to be the best among a total of eight different versions of his strategy. For the details of his strategy, see [1, 2].

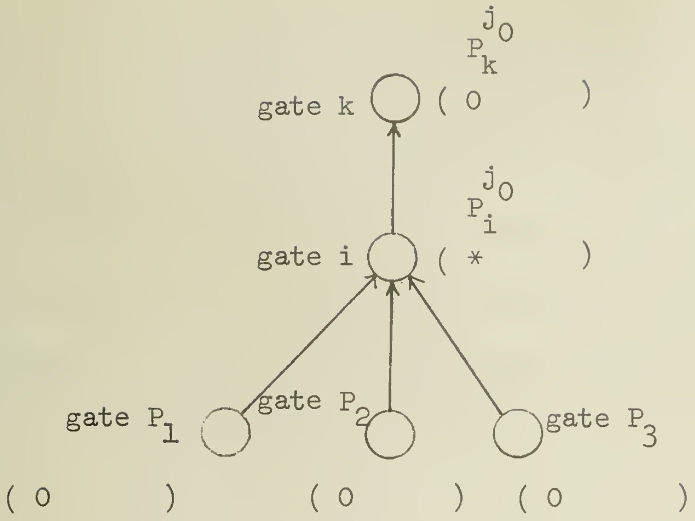


Fig. 2.1 An example of uncovered component $P_k^{j_0}$ of type 1-COV. $P_k^{j_0}$ can be covered by gate i by setting $P_k^{j_0}$ = 1, where setting $P_i^{j_0}$ = 1 does not introduce any uncovered components into the augmented intermediate solution.

In our branching strategy, we do not classify uncovered components having type G* possible covers into two categories as Davidson does. Thus, programming efforts and computation time which are spent on this classification in Davidson's approach are completely eliminated.

(ii) The selection criterion of uncovered components (SCUC).

Davidson utilizes the desirability order as the main concept of the SCUC. In order to improve the algorithm, he concentrates on ways of distinguishing uncovered components, by introducing several finer classifications of types of their possible covers. One of the most important concepts to obtain these finer classifications is the classification of of type VC* possible covers into 'bad' VC* possible covers, and 'good' VC* possible covers. A bad VC* possible cover for an uncovered component $P_k^{j_0}=0$ of gate k is an external variable x whose covering of $P_k^{j_0}$ (i.e., connecting x to gate k) results in a change of some type G* components in the immediatel succeeding gates to a less desirable type (Fig. 2.2). A possible cover of type VC* which is not 'bad' is called a 'good' VC* possible cover.

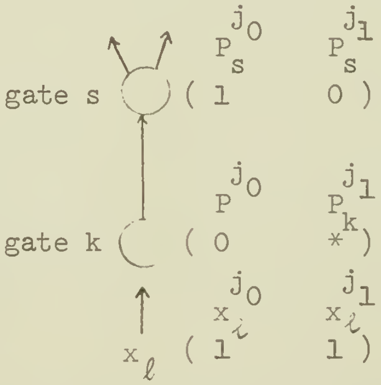


Fig. 2.2 An example of a 'bad' VC* possible cover $\mathbf{x}_\ell$. Suppose x_ℓ is a possible cover (of type VC*) for $P_k^{j_0}$. Before x_ℓ is connected to gate k, $P_s^{j_1}$ in gate s is of type G*. However, if x_ℓ is connected to gate k, the type of $P_s^{j_1}$ is changed to a less desirable type because $P_k^{j_1}$ becomes 0.

Based on this concept and several other properties of possible covers of types G* and VC*, he classified uncovered components of the original types G* and VC* into seven types:

G*/D, VC*G*/B', G*/C', VC*G*/A', G*/A', G*/B', VC*A

By combining the above classification of uncovered components, he obtains 'the selection order' as defined by:

G*/D-VC*G*/B'-G*/C'-VC*G*/A'-G*/A'-G*/B'-VC*/A-G*C*O*-G*C*O-GC*O*-GC*O,

where G*/D is the most desirable. (Definitions of uncovered components of types GC*O, GC*O*, G*C*O, and G*C*O* remain the same as ours.) This is the main part of his SCUC. According to this order, first he chooses a gate which contains uncovered components of the least desirable type. (If there are two or more such gates, he chooses the one with the smallest gate number in the case where the least desirable type is one of GC*O, GC*O*, G*C*O or G*C*O*, otherwise he chooses the one with the largest gate number.) If this gate has only one uncovered component of the least desirable type, then it must be chosen; but if this gate has two or more uncovered components of the same least desirable type, then he selects one which has the fewest possible covers.

In contrast to his SCUC, our SCUC employs the 'fewest possible covers' as the main criterion. By this criterion we select an uncovered component which has the fewest possible covers throughout the entire network. When a tie occurs, we apply 'the desirability order' to break it.

(iii) The implementation priority of possible covers (IPPC).

Suppose we are given the possible covers of an uncovered component which is selected from the current intermediate solution. Davidson determines the implementation priority of possible covers by using the lower bounds, $\tilde{C}_L$'s, of the costs corresponding to these possible covers, where $\tilde{C}_L$ for each possible cover is calculated as follows:

Let S denot e the current intermediate solution, and let $\hat{P}$ denote the uncovered component which is selected according to the SCUC.

Take a possible cover of $\hat{P}$, and implement it, generating the augmented intermediate solution S'.

If S' has type NWG components, then introduce appropriate new gates according to the scheme of 'Special treatment of type NWG components' described in Section 2.2. Rename the resulting intermediate solution as S'.

Then calculate $\tilde{C}_L$ using the formula

$$\tilde{C}_L = A \times \{\text{ the number of gates in S'}\}$$
$$+ B \times \{\text{ the number of connections in S'}\}$$
$$+ B \times \{\text{ the number of gates whose types are VC* or less desirable}\}.$$

If the possible cover under consideration is of type VC*, reduce $\tilde{C}_L$ by the amount of A + B, i.e., the modified $\tilde{C}_L$ = the original $\tilde{C}_L$ - (A + B).[†]

In our version, the implementation priority of possible covers is determined by a simplified desirability order,

G* - VC* - { GC*O, GC*O*, G*C*O, G*C*O*} - NWG

In summary, the difference between Davidson's branching strategy and our may be illustrated schematically in the following way. The entire searching

[+] This treatment has the purpose of assigning a higher priority to possible covers of type VC* than the original $\tilde{C}_L$ indicates.

process of the branch-and-bound algorithm is represented by a tree (called a search tree) in which the root node corresponds to the initial solution of the problem, and non-terminal nodes and terminal nodes correspond to intermediate solutions and feasible solutions, respectively. Davidson's branching strategy is based on minimizing the length of paths from the root node to terminal nodes. Therefore, the search tree corresponding to his strategy is shallow in depth, but it may become broad in width. On the other hand, our branching strategy is based on minimizing the number of of branches originating from each node. Therefore the entire search tree corresponding to our strategy is generally thinner in width.

3. COMPUTATIONAL RESULTS

The entire algorithm [3] was coded by incorporating the redundancy check and the branching strategy discussed in the previous sections. This computer program which is the modification of Davidson's algorithm with our branching strategy and the redundancy check procedure is called ILLOD-(NOR-B) [4].

In this section, we compare the above algorithm with Davidson's by solving problems of some single output functions and multiple output functions. Since his program was written in the CDC 1604 Assembler, we first implemented a FORTRAN program according to his strategy. We call this FORTRAN program the FORTRAN equivalent of Davidson's program. Column (a) of Table 3.1 shows part of his results taken from [1], and column (b) shows the corresponding results by the FORTRAN equivalent of Davidson's program. We note that both programs are not completely identical as the discrepancy in the number of backtracks indicates. But such a discrepancy seems unavoidable because of the following:

(i) Some of seven finely classified types G*/D, VC*G*/B', G*/C', VC*G*/A', G*/A', G*/B', VC*/A are not clearly defined in his text [1]. We had to guess the details to give a consistent definition for them.

(ii) In order to break ties in the SCUC as well as in the IPPC, we needed more detailed rules which are not written in the text. An example is the case where a gate has two or more uncovered components whose types are the least desirable and which have the fewest possible covers. (In the FORTRAN program, we choose one which is introduced most recently into the gate, if this case happens.) In the rest of this section, we refer to this FORTRAN equivalent of Davidson's program, instead of Davidson's CDC 1604 program.

Program / Problem	(a) Davidson's program (NANDECMP, for the CDC 1604)	(b) The FORTRAN equivalent of Davidson's program (IBM 360/75)	(c) The FORTRAN equivalent of Davidson's program with the redundancy check (I B M 360/75)	(d) ILLOD-(NOR-B) without redundancy check (I B M 360/75)	(e) ILLOD-(NOR-B) (IBM 360/75)
I. 77 functions of 3 or less variables for A = 100 and B = 1.	(total) 27 min 49 sec (total) 1910 BTKS	(total) 52.07 sec (total) 1081 BTKS	(total) 53.87 sec (total) 1074 BTKS	(total) 31.63 sec (total) 2146 BTKS	(total) 31.68 sec (total) 1895 BTKS
II. GIMPEL 1: a function of 4 variables $x_1x_4 \vee x_2\bar{x}_3x_4 \vee \bar{x}_2\bar{x}_3\bar{x}_4$ for A = 1 and B = 0. (The optimal solution has 6 gates).	15 sec 10 BTKS	0.88 sec 6 BTKS	0.89 sec 6 BTKS	0.82 sec 20 BTKS	1.62 sec 19 BTKS
III. GIMPEL 1: for A = 100 and B = 1. (The optimal solution has 6 gates/ 13 connections).	52 sec 54 BTKS	3.09 sec 43 BTKS	2.78 sec 35 BTKS	4.21 sec 278 BTKS	3.59 sec 176 BTKS
IV. GIMPEL 2: a function of four variables $x_1\bar{x}_2\bar{x}_3 \vee \bar{x}_1x_2\bar{x}_3 \vee \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_2\bar{x}_4$ for A = 1 and B = 0. (The optimal solution has 7 gates).	16 sec 19 BTKS	1.77 sec 21 BTKS	1.63 sec 21 BTKS	1.37 sec 48 BTKS	1.71 sec 48 BTKS
V. GIMPEL 2: for A=100 and B=1. (The optimal solution has 7 gates/19 connections).	2 min 11 sec 138 BTKS	25.50 sec 450 BTKS	26.82 sec 450 BTKS	6.36 sec 324 BTKS	6.68 sec 324 BTKS

VI. PRATHER: a network of 5 functions of 3 variables for A=1 and B=0. (The optimal solution has 11 gates).	34 sec 33 BTKS	4.31 sec 40 BTKS	3.61 sec 40 BTKS	0.88 sec 34 BTKS	1.07 sec 34 BTKS
VII. PRATHER: for A = 100 and B = 1. (The optimal solution has 11 gates/ 20 connections).	11 min 36 sec 750 BTKS	69.03 sec 786 BTKS	67.37 sec 786 BTKS	12.07 sec 729 BTKS	12.70 sec 729 BTKS
VIII S-AND-D: a network of 4 functions of 5 variables for A=1 and B=0. (The optimal solution has 8 gates).	4 sec 3 BTKS	1.33 sec 3 BTKS	1.38 sec 3 BTKS	0.64 sec 4 BTKS	1.00 sec 4 BTKS
IX. S-AND-D: for A = 100 and B = 1. (The optimal solution has 8 gates/ 17 connections).	2 min 10 sec about 150 BTKS	16.51 sec 71 BTKS	16.29 sec 70 BTKS	6.82 sec 127 BTKS	6.93 sec 126 BTKS
X. One-bit adder: for A=1 and B=0. (The optimal solution has 8 gates).	32 sec 42 BTKS	2.24 sec 38 BTKS	2.43 sec 38 BTKS	1.63 sec 84 BTKS	2.09 sec 54 BTKS
XI. One-bit adder: for A=100 and B = 1. (The optimal solution has 8 gates/23 connections).	(unfinished in 6 min 58 sec)	18.90 sec 333 BTKS	17.74 sec 333 BTKS	9.17 sec 704 BTKS	6.34 sec 379 BTKS

TABLE 3.1 Statistics of the computation times and the numbers of backtracks for the problems taken from the text [1].

Notes for preceding Table 3.1.

Note 1. All problems are single optimum problems, i.e., the program searches for only one optimal network for each problem. No fan-in or fan-out restriction is imposed.

Note 2. A = 100 and B = 1 means to minimize the number of gates primarily, then to minimize the number of connections secondarily, and A = 1 and B = 0 means to minimize the number of gates only.

Note 3. Columns (b), (c), (d) and (e) are the results for the problems for dual functions of Davidson's, since these programs are for NOR networks, while (a) Davidson's program is for NAND networks.

It should be worthwhile also to notice the difference of the languages used for both programs. For the logical design program by the branch-and-bound algorithm, coding by machine instructions seems to have an apparent advantage over the use of FORTRAN in terms of the efficiency of the program. For example, by using machine instructions, the output of each gate can be stored in a few (machine) words in which each bit represents the output component for an input vector. Access to the output of a gate during the computation (this is one of the operations most frequently used in the program) can be performed by a single instruction. However, in the FORTRAN program, each output component of a gate occupies one word. Therefore, the equivalent operation is completed by a repeated access to the set of words. (The more the external variables of the problem, the more the difference of the access method may affect the efficiency of the program.)

Before running ILLOD-(NOR-B), we experimented two more programs, the one is the FORTRAN equivalent of Davidson's program with the redundancy check procedure, and ILLOD-(NOR-B) without the redundancy check procedure. Columns (c) and (d) show the results by these two programs. Finally, we ran ILLOD-(NOR-B). The result is shown in column (e).

Investigating the results shown in columns (b), (c), (d) and (e), let us discuss the difference of the programs in the two aspects, i.e., (i) the difference in the branching strategy, and (ii) the gimmick of the redundancy check.

(i) In the first aspect, we compare the computation time by the FORTRAN equivalent of Davidson's program (column (b)) with the one by ILLOD-(NOR-B) without the redundancy check procedure (column (d)). The result is that the latter program is faster than the former one, except for one problem (Problem III). The improvement would be greater for more gates. The number of backtracks counts generally more by the latter program than by the former one. This is due to the difference in counting backtracks. We must reduce the figures by the latter program by about 50% if we want to compare them with the corresponding figures by the former program for the following reason. In the ILLOD-(NOR-B) without the redundancy check procedure, the cost lower bound $\tilde{C}$ corresponding to a possible cover is obtained from the cost estimate of the current intermediate solution plus the cost estimate of possible cover under consideration.

On the other hand, in Davidson's program, the cost lower bound $\tilde{C}_L$ is obtained from the cost estimate of the augmented intermediate solution corresponding to the possible cover under consideration. Clearly $\tilde{C}_L$ appears to be not less than $\tilde{C}$ if both possible covers are identical. In both cases, by the backtrack procedure, we search for an unimplemented possible cover whose cost lower bound does not exceed the cost ceiling $\bar{C}$. When we encounter such a possible cover, we increment the number of backtracks by 1. Due to the above difference of the value of the cost lower bound, the number of backtracks by ILLOD-(NOR-B) should be not smaller than by Davidson's program, assuming that the search trees by both programs are identical. If we want to make the counting scheme of ILLOD-(NOR-B) identical to Davidson's, we count 1 only when we obtain a new augmented intermediate solution whose estimated cost does not exceed $\bar{C}$. In ILLOD-(NOR-B) even if we count 1, the corresponding augmented solution may exceed the $\bar{C}$ and be immediately discarded.

By this change of the counting scheme, the average number of backtracks for sample problems appears about 50% fewer, according to the experiment. (The way of counting backtracks has nothing to do with the efficiency of the program at all.)

(ii) In the second aspect, i.e., the effect of the redundancy check on the improvement of computation speed, we compare the results in columns (b) and (c), and also the results in columns (d) and (e). For some problems, the gimmick of the redundancy check reduces the computation time, but for some other problems, this gimmick does not. Clearly, if the first feasible solution happens to be optimal, this gimmick does not reduce the computation time, but rather adds a slight amount of running time spent for redundancy check; or, even if the first feasible solution is not optimal, this gimmick will not reduce the computation time if the total amount of time for the problem is very small. For many of the above test problems, these situations happen, thus the use of the redundancy check procedure is not well justified from the above results.

In order to experiment the algorithms further, we took some 6-gate and 7-gate functions of four variables. Table 3.2 is the statistics of these functions, by using the above four FORTRAN programs. For 6-gate functions of four variables, ILLOD-(NOR-B) is slightly worse than ILLOD-(NOR-B) without the redundancy check procedure, even though the number of backtracks is reduced by about 10%. But, for 7-gate functions of four variables, the redundancy check procedure reduces about 20% of computation time from 20.33 seconds to 16.37 seconds. Thus, the gimmick of the redundancy check seems to improve the computation speed for those problems which require many gates. Tables 3.3 ~ 3.6 are the detailed statistics by the four FORTRAN programs, showing the computation time vs. the number of true vectors. These figures are plotted in Fig. 3.1 for 6-gate functions, and Fig. 3.2 for 7-gate functions.

One thing should be worthwhile to mention about the statistics:

The number of functions experimented are 83 for 6-gate functions, and 46 for 7-gate functions. But the FORTRAN equivalent of Davidson's program did not terminate the computation for one function among 46 7-gate functions in 700 seconds. (The best feasible solution found so far still contains 9-gates and 18 connections.) In order to take the average of computation times, therefore, we excluded this function, which resulted in the statistics of 45 7-gate functions.

Program / Problem	The FORTRAN equivalent of Davidson's program (IBM 360/75)	The FORTRAN equivalent of Davidson's program with the redundancy check procedure (I B M 360/75)	ILLOD-(NOR-B) without the redundancy check procedure	ILLOD-(NOR-B) (IBM 360/75)
I. 83 6-gate functions of 4 variables.	4.57 sec 69.7 BTKS	4.66 sec 67.4 BTKS	3.00 sec 216.7 BTKS	3.15 sec 191.8 BTKS
II. 45 7-gate functions of 4 variables.	31.65 sec 518 BTKS	29.78 sec 470 BTKS	20.33 sec 1518 BTKS	16.37 sec 1148 BTKS

TABLE 3.2 Statistics of the computation times and the number of backtracks for sample functions of 4 variables.

Note 1. All problems are multiple optimum problems, i.e., the program searches for all equivalent optimal networks for each problem.

Note 2. The optimal networks have the minimum number of inputs to gates among those which have the minimum number of gates, (i.e., the case of A=100 and B=1).

Note 3. The figures are the average per function.

Number of gates	Number of true vectors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	Number of functions		1	2	3	8	11	5	6	18	12	2	10	2	2	1
	Average computation time		1.18 sec	2.93	6.54	8.08	7.25	3.35	4.30	3.96	2.65	10.66	2.76	5.32	1.12	0.32
	Average number of backtracks		21 BTKS	42	113.7	130.1	113.7	43.2	66.8	56.7	40.1	157.0	42.7	77.5	15.0	4.0
7	Number of functions		2	2	4	3*	4	4	4	4	4	4	4	4	2	
	Average computation time		34.69 sec	44.86	73.80	14.7	33.44	46.55	50.55	19.29	13.21	25.67	24.53	17.39	1.63	
	Average number of backtracks		642.0 BTKS	697.5	1203.8	239.7	567.5	795.0	786.3	320.8	194.8	409.8	404.3	282.0	28.5	

TABLE 3.3 Computation time vs. number of true vectors for 83 6-gate functions and 45 7-gate functions of four variables; by the FORTRAN equivalent of Davidson's program.

* Originally we experimented four functions. But the FORTRAN equivalent of Davidson's program did not terminate the computation for one function out of four, in 700 seconds. We excluded this result from the statistics.

Number of gates	Number of true vectors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	Number of functions		1	2	3	8	11	5	6	18	12	2	10	2	2	1
	Average computation time		1.30 sec	2.70	6.44	8.45	7.49	3.49	4.28	4.03	2.68	10.62	2.85	5.34	1.13	0.31
	Average number of backtracks		21.0 BTKS	39.0	109.3	126.5	111.0	42.8	64.8	54.6	37.3	144.0	42.2	77.5	15.0	4
7	Number of functions		2	2	4	3*	4	4	4	4	4	4	4	4	2	
	Average computation time		32.01 sec	43.12	67.72	14.25	27.42	40.31	50.5	19.78	13.80	26.34	21.54	18.43	1.77	
	Average number of backtracks		633.5 BTKS	684.0	1196.8	235.3	455.3	459.5	770.8	320.8	194.0	409.0	350.8	282.0	28.5	

TABLE 3.4 Computation time vs. number of true vectors for 83 6-gate functions and 45 7-gate functions of four variables; by the FORTRAN equivalent of Davidson's program <u>with</u> the redundancy check procedure.

* See the note Table 3.3, P. 49

Number of gates	Number of true vectors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	Number of functions		1	2	3	8	11	5	6	18	12	2	10	2	2	1
	Average computation time		0.99 sec	2.63	3.02	7.71	4.72	2.47	3.63	2.35	1.92	1.37	1.47	1.04	0.33	0.28
	Average number of backtracks		62 BTKS	144.5	181.3	615.3	384.5	167.4	290.8	169.1	99.7	66.0	83.0	42.5	20.5	28
7	Number of functions		2	2	4	3*	4	4	4	4	4	4	4	4	2	
	Average computation time		37.82 sec	19.74	29.75	17.91	50.34	16.80	29.39	17.83	6.81	8.42	22.43	3.53	2.43	
	Average number of backtracks		3108.0 BTKS	1444.0	2181.8	1659.7	3670.8	1103.8	2085.0	1048.3	491.0	499.8	2096.0	234.5	297.5	

TABLE 3.5 Computation time vs. number of true vectors for 83 6-gate functions and 45 7-gate functions of four variables; by ILLOD-(NOR-B) without the redundancy check procedure.

* See footnote in TABLE 3.3, p. 49

Number of gates	Number of true vectors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	Number of functions		1	2	3	8	11	5	6	18	12	2	10	2	2	1
	Average computation time		1.00 sec	1.65	3.26	8.55	5.05	2.44	3.99	2.12	2.04	1.74	1.74	1.43	0.43	0.29
	Average number of backtracks		50 BTKS	80.5	173.0	593.6	377.7	140.2	280.3	102.2	85.3	61.5	76.2	42.5	20.5	28.0
7	Number of functions		2	2	4	3*	4	4	4	4	4	4	4	4	2	
	Average computation time		31.38 sec	16.46	26.35	17.44	39.11	15.53	26.21	12.14	6.66	8.87	7.46	4.03	1.73	
	Average number of backtracks		2674.5 BTKS	1245.5	1941.3	1574.0	2760.3	1002.5	1779.0	620.3	436.3	479.3	451.3	232.5	147.5	

TABLE 3.6 Computation time vs. number of true vectors for 83 6-gate functions and 45 7-gate functions of four variables; by ILLOD-(NOR-B).

* See footnote Table 3.3, p.49.

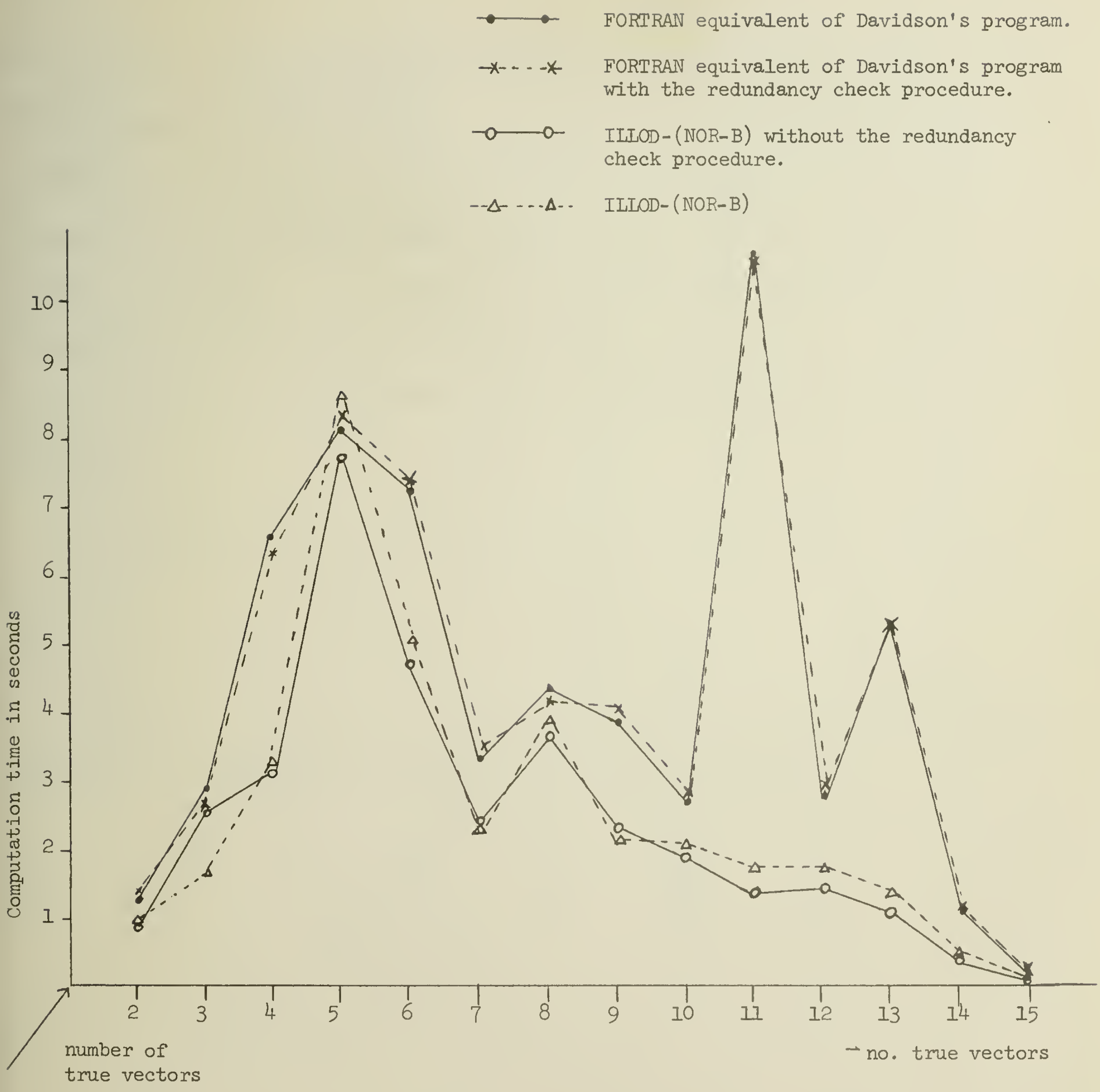


Fig. 3.1 Computation time vs. number of true vectors for 6-gate functions of four variables.

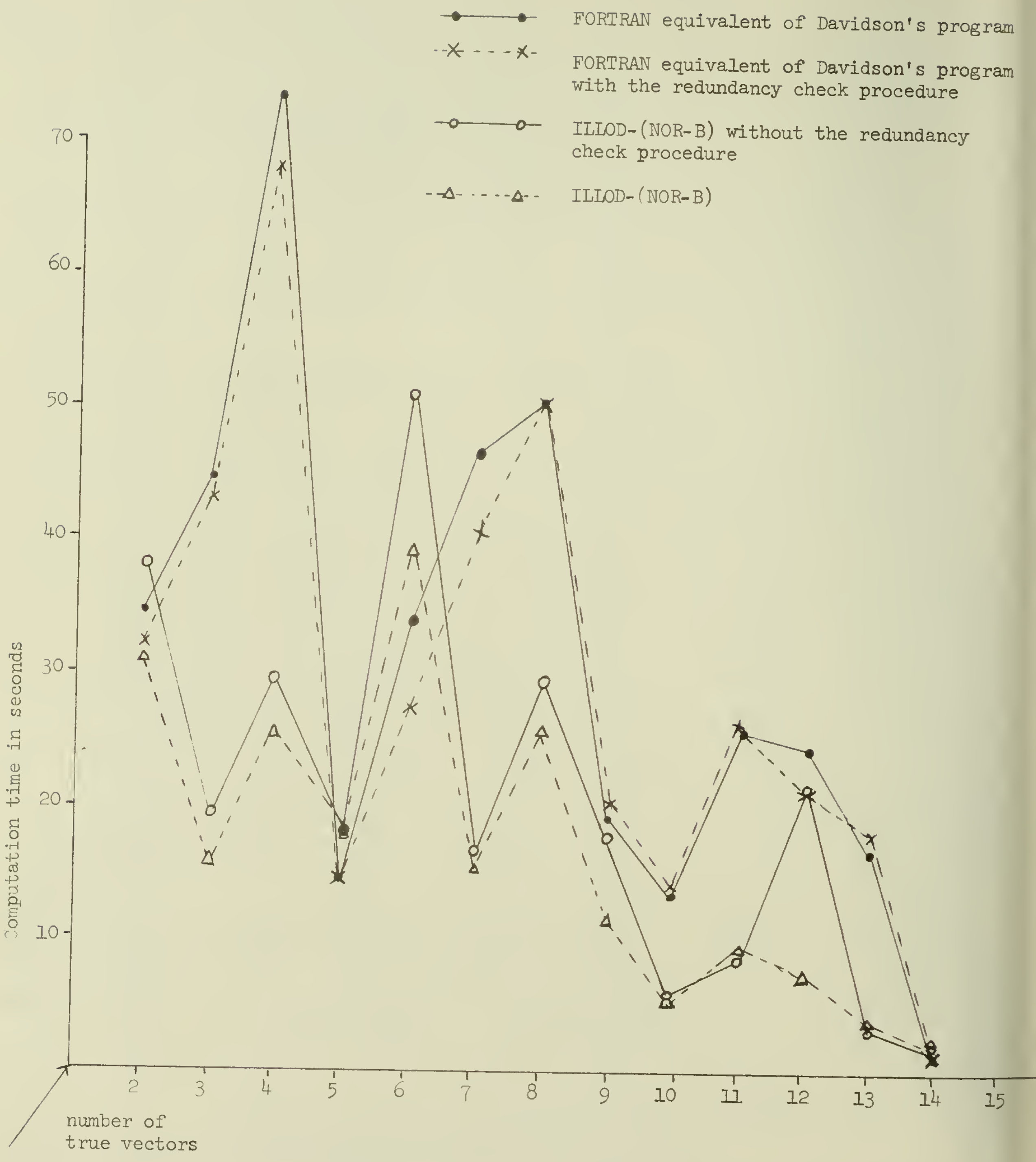


Fig. 3.2 Computation time vs. number of true vectors for 7-gate functions of four variables.

Finally we experimented functions of four variables which require 8 or more gates. Since these functions are more time-consuming than the previous ones we tested only a limited number of functions so far, and attempted to solve mainly by using ILLOD-(NOR-B). Table 3.7 shows the results. In Table 3.7, each function is denoted by the hexadecimal equivalent of a 16-bit binary representation $(a_0, a_1, \ldots, a_{15})$, where a_i, i=0, ..., 15, are the coefficients of the minterm expansion of the function f:

$$f = a_0 \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 \vee a_1 \bar{x}_1 \bar{x}_2 \bar{x}_3 x_4 \vee a_2 \bar{x}_1 \bar{x}_2 x_3 \bar{x}_4 \vee a_3 \bar{x}_1 \bar{x}_2 x_3 x_4 \vee \ldots \vee a_{15} x_1 x_2 x_3 x_4 .$$

Because of a limited number of test functions, we did not take the average of the computation times or of the number of backtracks.

Table 3.7

Follows

Number of true vectors	Function HEX No.	The FORTRAN equivalent of Davidson's program			The FORTRAN equivalent of Davidson's program with the redundancy check procedure			ILLOD-(NOR-B) without the redundancy check procedure			ILLOD-(NOR-B)		
		First Sol. → Opt. Sol.	Computation Time (Sec.)	No. of BTKS	First Sol. → Opt. Sol.	Computation Time (Sec.)	No. of BTKS	First Sol. → Opt. Sol.	Computation Time (Sec.)	No. of BTKS	First Sol. → Opt. Sol.	Computation Time (Sec.)	No. of BTKS
3	0016	9/20 → 8/17	20.29	327	9/20 → 8/17	19.82	321	9/24 → 8/17	46.66	3587	9/24 → 8/17	35.46	2574
	0118										10/27 → 8/19	88.12	6876
4	0069	8/22 → 8/22	59.51	1035	8/22 → 8/22	58.68	1035	12/31 → 8/22	79.37	57.09	12/31 → 8/22	51.12	3188
	0116										12/37 → 9/23	189.64	13347
	2884										10/26 → 8/19	1121.85	84904
5	009E	8/19 → 8/17	46.94	722	8/19 → 8/17	46.12	718	9/22 → 8/17	57.07	4455	9/22 → 8/17	40.35	3076
	1990										8/20 → 8/16	141.17	9956
	0998										8/21 → 8/16	66.21	4641
6	013E	10/21 → 8/18	7.74	97	10/21 → 8/18	7.50	97	10/29 → 8/18	25.99	1883	10/29 → 8/18	16.34	1116
	29C2										12/37→(9/22)	Not finished within 2700	more than 184951
	0AD4										9/23 → 8/16	59.35	4304
7	017E	10/22 → 8/19	13.85	211	10/22 → 8/19	13.48	211	10/30 → 8/19	93.14	6629	10/30 → 8/19	30.39	2225
	1AE1										12/31 → 9/21	1859.14	100258
	88B6										10/27 → 8/22	64.91	4613
8	03TE	9/19 → 8/18	13.24	197	9/18 → 8/18	12.54	197	9/26 → 8/18	33.56	2515	9/26 → 8/18	22.08	1660
	3CC9										12/32 → 9/18	1449.48	7913
	1AB9										8/23 → 8/21	17.61	1049

9	[illegible]	9/[illegible] → 8/18	53.6[illegible]	770	9/[illegible] → 8/18	[illegible]	[illegible]	9/27 → 8/18	36.63	2836	9/27 → 8/18	4.51	1837
	8[illegible]B										10/[illegible] → 8/[illegible]	111.46	6486
	A[illegible]										[illegible] → 8/[illegible]	45.45	2823
10	EEE1	9/20 → 8/16	354.01	5418	9/20 → 8/16	351.26	5418	11/23 → 8/16	95.94	5185	11/[illegible] → 8/16	96.28	5104
	967E										9/31 → 8/[illegible]	86.73	5170
	9ABE										9/24 → 8/18	59.75	4197
11	FEE8				8/22 → 8/19	more than 790		8/19	94.87	6074	8/19	93.83	6074
	BCDE										9/25 → 8/19	39.99	2295
	E8FE										9/20 → 8/16	50.55	4647
12	FEE9				12/31	more than 300					11/29 → 9/24	239.82	11601
	9BFE										11/27 → 8/18	20.05	1332
	6BFE										12/29 → 8/21	624.33	59226
13	FEEB					more than 60		10/23 → 8/19	12.71	673	10/23 → 8/19	12.44	673
	EFF6										9/21 → 8/18	11.96	820
	EFF9				9/24 → 8/19	220.94	3491	9/24 → 8/19	18.70	963	9/24 → 8/19	19.07	963
	EBFE				8/19 → 8/18	14.73	184	9/20 → 8/18	9.38	674	9/20 → 8/18	10.38	665

TABLE 3.7 Computation time vs. number of backtracks for functions of four variables which require 8 or more gates.

REFERENCES

1. E. S. Davidson, "An algorithm for NAND decomposition of combinational switching functions," Ph.D dissertation, Department of Electrical Engineering and Coordinated Science Laboratory, University of Illinois, 1968.

2. T. Nakagawa and S. Muroga, "Exposition of Davidson's thesis 'An algorithm for NAND decomposition of combinational switching systems'," to be published, Department of Computer Science, University of Illinois.

3. T. Nakagawa and H. C. Lai, "A branch-and-bound algorithm for optimal NOR networks (The algorithm desceiption)," Report No. 438, Department of Computer Science, University of Illinois.

4. T. Nakagawa and H. C. Lai, "Reference manual of FORTRAN program ILLOD-(NOR-B) for optimal NOR networks," to be published as a report, Department of Computer Science, University of Illinois.

5. T. Nakagawa, "A branch-and-bound algorithm for optimal AND-OR networks (The algorithm description)," to be published as a report, Department of Computer Science, University of Illinois.

6. T. Nakagawa, "Reference manual of FORTRAN program ILLOD-(AND-OR-B) for optimal AND-OR networks," to be published as a report, Department of Computer Science, University of Illinois.

7. S. W. Golomb and L. D. Baumert, "Backtrack programming," Journal of the Association for Computer Machinery, Vol. 12, no. 4, pp. 516-524, October 1965.

8. C. R. Baugh, T. Ibaraki, T. K. Liu and S. Muroga, "Optimum network design using NOR and NOR-AND gates by integer programming," Report No. 293, Department of Computer Science, University of Illinois.